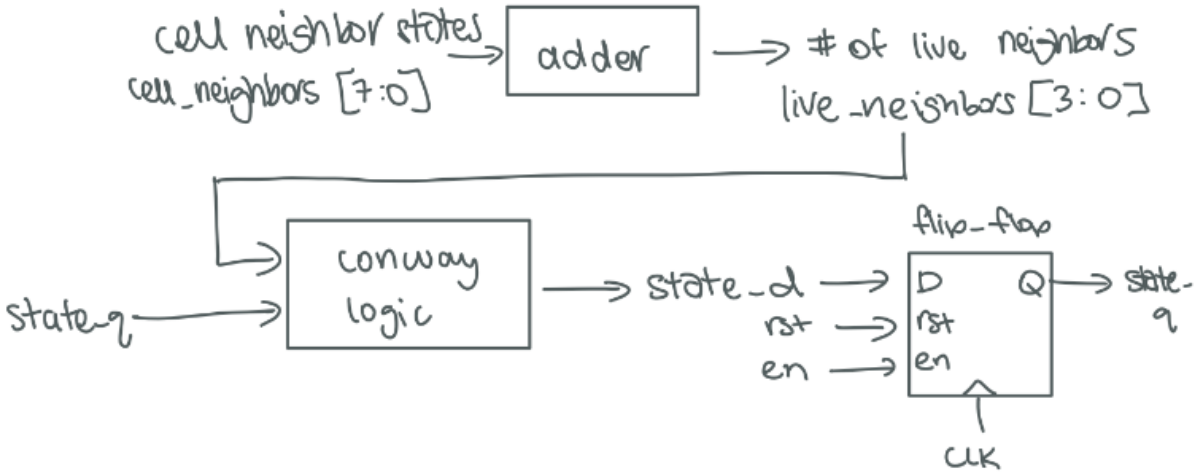
Lab 1: Conway's Game of Life

Cara Mulrooney and Dasha Chadiuk

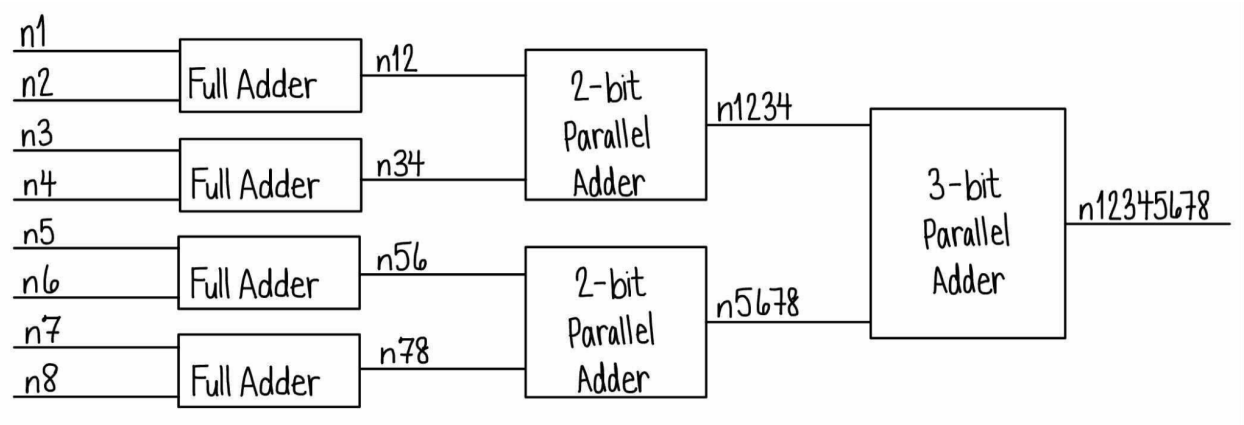
Conway's Game of Life Architecture

Our system consists of a module of cells, where each cell is represented by a D flip-flop. For each cell, we follow the same combinational logic to determine its next state. First, we analyze the states of all eight neighboring cells, a binary value of either a 0 or 1, around the particular cell. Next, we feed these bit values into an 8-bit adder to calculate the sum of "live" neighboring cells, which have a binary values of 1. Then, we compare this sum with the rules of Conway's Game of Life to determine the next state for the particular cell we are analyzing. This state data is sent as an input, state_d, into the D flip-flop representing the cell. The next state of the cell is the output of the D flip-flop, state_q. A block diagram of our system architecture is shown below.



Designing Our 8-bit Adder

To construct our 8-bit adder, we first grouped the bits into pairs to be fed as inputs into four 1-bit full adders. Full adders are combinational circuits that perform the arithmetic sum of three bits: A, B, and C_in (input carry). This third input, C-in, typically represents the extra bit that is generated every time a previous addition of two N-digit numbers results in a sum with N+1 digits. Here, it is set as 1'b0 for all full adder calculations at this step, since we aim to find all neighboring cells that are already "on" or have a binary value of 1. The result of four 1-bit full adders is four 2-bit numbers, where each resulting bitstring represents the number of live neighbors in one-fourth of the original eight surrounding cells. These outputs are then treated as inputs to two 2-bit ripple-carry adders, where each ripple-carry adder utilizes two full adders to sum two 2-bit binary numbers. This step has an output of two 4-bit numbers, where each resulting bitstring represents the sum of the live neighbors in one-half of the original eight surrounding cells. Finally, using the outputs of the 2-bit adders as inputs, we employ one 3-bit ripple carry adder to calculate the total sum of all live neighboring cells.



Conway Cell Logic

The following truth table represents the relationship between the total sum of live neighboring cells, the output of our 8-bit adder, and the resultant state of the D flip-flop that represents each cell. Please note that the rows with a sum total of two and three live neighbors are highlighted, as these conditions yield a result state of the previous state of the cell and 1, respectively.

#of live neighbors	adder output live[3]	live[2]	live[1]	live[0]	state_q result state
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	prev_state
3	0	0	1	1	1
4	0	1	0	0	0
5	0	1	0	1	0
6	0	1	1	0	0
7	0	1	1	1	0
8	1	0	0	0	0

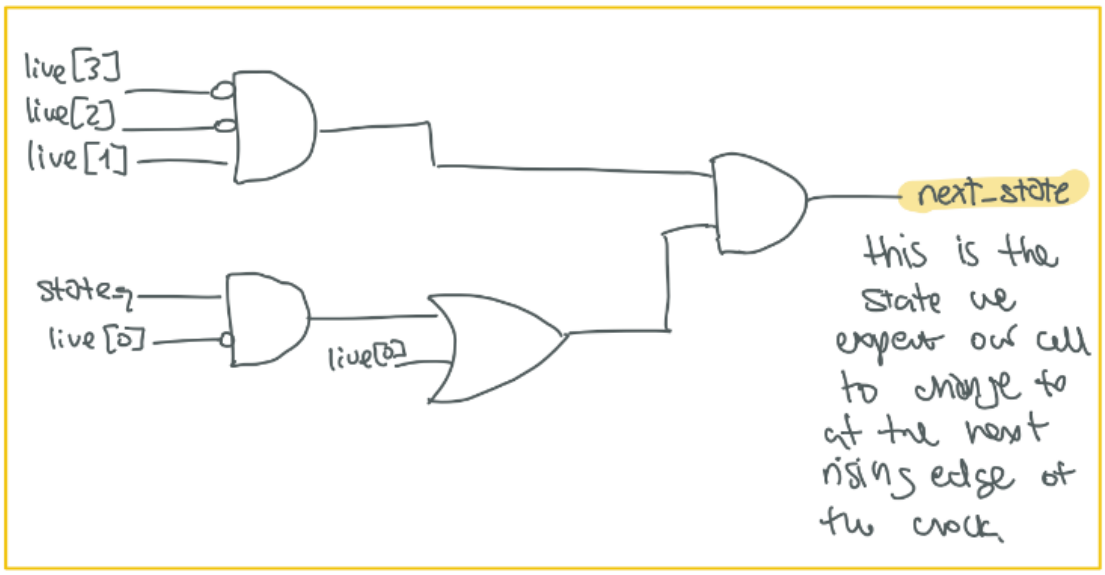
As we can note from the above truth table, state_d will be 1 if the cell has three neighbors that are alive OR if its previous state was 1 and it has two live neighbors. Three of the four outputs of the 8-bit adder (live[1], live[2], and live[3]) are the same for both of these situations. So, we can construct the following circuit to represent a part of the output for both situations.

However, we still need to represent both of these situations with logic utilizing live[0] and state_q, since the output of a total sum of two live neighbors depends on the previous state. Either of the following scenarios should produce an output of 1:

1. When a cell has two live neighboring cells, state_q=1, and live[0]=0
2. When a cell has three live neighboring cells and live[0]=1

So, taking live[0] and state_q into consideration, we can employ the following logic diagram to represent both situations. Please note that both cases cannot be true at the same time, so we used an "or" gate to find the output for this section of our Conway logic.

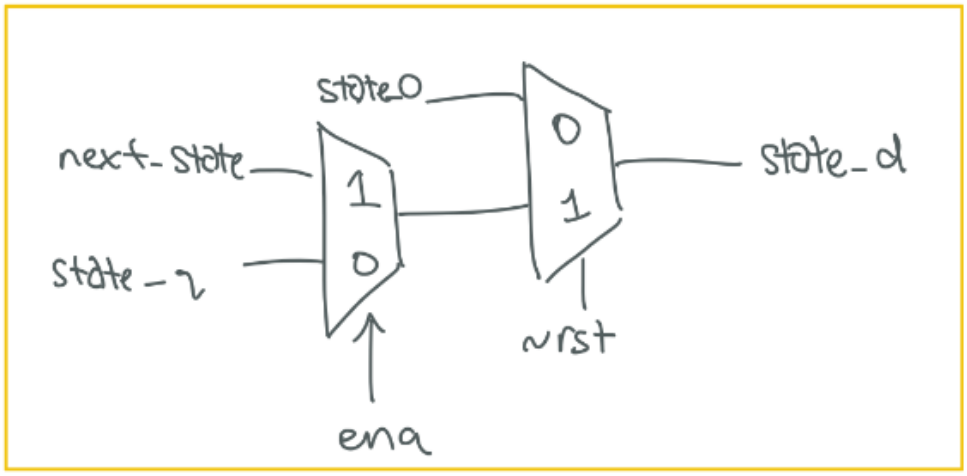
Putting it all together, the state of the cell should have a binary value of 1 if the "and" logic gate output, mentioned in the first part of this Conway logic section, has a binary value of 1 and if the "or" logic gate, mentioned in the second part of this section, also has a binary value of 1.



As shown in the logic diagram above, "next_state" represents the state we expect our cell to change to at the next rising edge of the clock.

When the cell changes state, we also need to check the status of the reset and enable pins of the

D flip-flop. To make this observation, we can use two muxes, in a configuration shown below.

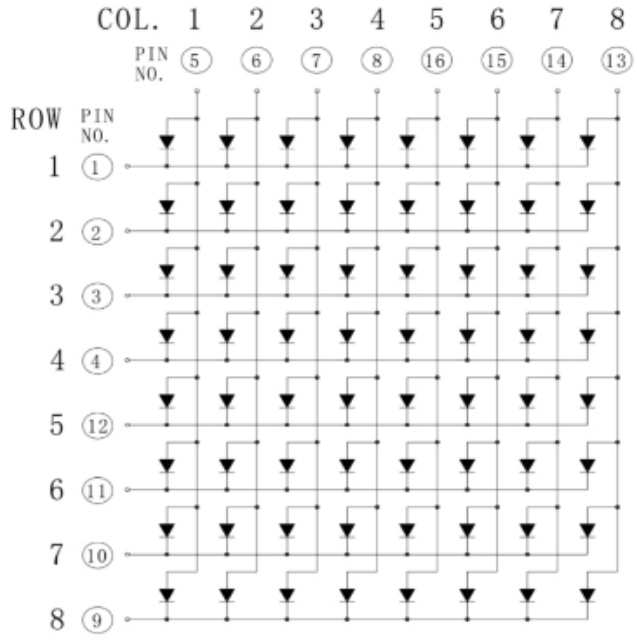


When enable, ena, has a binary value of 1, we want to set the cell state to the one computed using Conway logic. Similarly, we want to continue to set the state of the cell while ~reset, ~rst, has a binary value of 0. When ~rst has a value of 1, we want the cell to return to its original state, state_0.

As we previously mentioned, each cell is represented by a D flip-flip, where the output, state_q, follows state_d on the rising edge of the clock.

Led Driver

The overall layout of our 8x8 LED Display is shown below.



Here, the cells of each row can be represented using the following notation, where N^2 bits is the state of each cell in the grid, 1 is a live cell (LED is “on”), and 0 is a dead cell (LED is “off”).

cells [N*N-1:0]

Each column is selected using a 3 to 8 decoder and each row is chosen using “nor” logic. So, when we drive the LEDs, the columns are pulled high and the rows are pulled low. The logic for this part of our system follows the idea that an entire row is pulled low if any cell is alive from that row. Each row can be represented using the following notation.

row[0] = ~|cells[N-1:0]

row[1] = ~|cells[2x(N-1):N]

row[2] = ~|cells[3x(N-1):2N]

row[3] = ~|cells[4x(N-1):3N]

can be rewritten as 3N+(N-1)

The representation above can be consolidated to generalize an expression for all rows, shown below.

row[N] = ~|cells[(N*row)+(N-1):N*row]